Secure Login

Stronger Authentication With Keystroke Features

October 19, 2014

Manish Choudhary(GTID: 902982487) Anusha Gargi(GTID: 903087876)

Introduction

SecLogin is a more secure authentication system as compared to traditional “password-only” alternatives. SecLogin enforces two factor authentication providing more entropy and thus, more security. In this approach, the legitimate user's password is combined with the user's typing patterns (keystrokes and latencies between keystrokes) to facilitate the use of a hardened password to authenticate login request. This hardened password is protected using Shamir’s secret sharing scheme and thus, is more secure than the conventional password alone.

System Design

The authentication system implements following key features in the order of execution:

1. Takes username as input and checks whether the user is registered or not.
2. If the user is not registered, asks if he wants to, gets the password, and performs registration and initialization
   - Selects a random 160-bit prime number q for this user which would define the group.
   - Selects a random hardened password “hpwd” in the range (0,q).
   - The number of feature vectors (h) to be allowed in history file is set to 8.
   - Generates 14 random coefficients in the range (0,q) to form a polynomial with hpwd as the constant value in the polynomial.
   - Generates 15 alpha and 15 beta values by using the formulas as discussed in [1]. Y values for alpha are generated by putting X=2i for i = [1,15] and Y values for beta are generated by putting X=2i+1 for i = [1,15]. HMAC with SHA 1 is used as the keyed hash function.
   - The instruction table is stored to the non-volatile memory so that it can be used to authenticate next login request.

- A constant fixed size history file is create. In the current implemented, it's size is fixed to 1048 bytes and it's encrypted version(which is stored to the hard drive after encrypting with hpwd) is 1056 bytes in size always. Success identifier is added to the user history to verify the correct decryption at the time of login request. AES 128 bit CBC mode is used to encrypt the history file before storing is to the disk.

3. If the user is already registered, asks the user to enter password for authentication.
4. Two modes of execution are provided, normal mode and test mode.
5. In normal mode, user is asked to enter the 15 feature values through terminal and the authentication of login request takes place.
6. Test mode is provided for testing and grading purpose. In test mode, sample file with multiple feature vectors can be provided to execute multiple login requests at once. The sample file should be of this format:

<value 1> <value 2> .....<value 15>
<value 1> <value 2> .....<value 15>
<value 1> <value 2> .....<value 15>
.
.

7. Another feature "debugMode" is provided in test mode. If enabled, intermediate results are printed to facilitate analysis and verification of authentication system.
8. The login request is authenticated in following steps:
   - q, alpha[] and beta[] values are loaded in the memory from the file stored on the disk
   - 15 (X, Y) pairs are recovered using q, alpha, beta, input password, input feature values and the threshold values based on the criteria described in [1]. For each pair, alpha or beta is selected based upon the relation of input feature value to the threshold.
   - The hardened password hpwd' is recovered using polynomial interpolation.
   - This recovered hpwd' is used to decrypt the history file and the decryption is verified by looking for the success identifier in the decrypted user history.
   - If the decryption fails, the service terminated otherwise following post authentication operations are performed.
9. On successful login, following operations are performed:
   - The new feature vector (for the current login) is added to the history file(removing the oldest one in case the file already contains 8 feature vectors) and the encrypted history file is stored back to the disk ensuring that its size remains constant.

- Keeping q same, new hpwd, polynomial and thus, new alpha and beta values are generated.
- After first 8 successful logins, mean and standard deviation are also calculated using the features vectors stored in the user history file to decide which of the alpha and beta values need to be garbled for the distinguishing features. Correct values are kept for both alpha and beta in case of non-distinguishing features.
- q, alpha[] and beta[] are stored back to the disk to facilitate authentication of next login request.

Threshold vector and k are system parameters defined as global variables with can be tuned as per the security and usability requirements.

Implementation

Following functions have been implemented to develop the authentication service.

❖ **HMACGenerator()**

unsigned char * HMACGenerator(const unsigned char * message, const char * password)

This function takes in a message and password as the parameters to produce a keyed hash. SHA1 is the hash function used to compute HMAC. HMAC function is provided in openssl's crypto library.

❖ **strip_newline()**

void strip_newline(char * str, int size)

Helper function to remove newline or NULL characters from the character string passed as argument.

❖ **int_array_to_string()**

string int_array_to_string(int int_array[], int size_of_array)

This function takes an array and its size as the parameters to covert the integer array as an argument to a c++ string.

- **EVPInitialize()**

int EVPInitialize(unsigned char * password, int passwordLength, unsigned char * salt, EVP_CIPHER_CTX * enCTX, EVP_CIPHER_CTX * deCTX)

Function to initialize the EVP encryption/decryption. It is a generic function which can be used to initialize any encryption/decryption mode. It must be called before calling encrypt/decrypt functions. Currently, it takes password and salt to generate the key and IV for aes_128_cbc mode. EVP is a generic wrapper provided in openssl crypto lib to facilitate a single interface to different underlying encryption/decryption modes.

- **encrypt()**

unsigned char * encrypt(unsigned char * plaintext, int plaintextLength, EVP_CIPHER_CTX *ctx, int *cipherLength)

This function takes plain text and plaintext length to convert it into a cipher text with the given cipher text length using the mode, key and IV set in the EVP_CIPHER_CTX via call to EVPInitialize().

- **decrypt()**

unsigned char * decrypt(unsigned char * ciphertext, int cipherLength, EVP_CIPHER_CTX * ctx, int * plaintextLength)

This function decrypts the ciphertext using the mode, key and IV set in EVP_CIPHER_CTX via call to EVPInitialize().

- **generateAlphaBeta()**

int generateAlphaBeta(BIGNUM * alpha[], BIGNUM * beta[], BIGNUM * q, BIGNUM * hpwd, const char * password)

The function takes in q, hpwd and password as input and generates 14 random coefficients in the range(0,q). A polynomial is generated using the hpwd and these 14 coefficients. A set of 15 (X,Y) pairs are generated using the polynomial by setting X = 2i for i [1, 15] and another set of 15 (X,Y) pairs are generated by setting X = 2i+1 for i [1, 15] .15 Alpha values are generated by adding the Y values to the HMAC of corresponding X which is a module q operation.Generate 15 Beta values by adding the Y values to the HMAC of corresponding X. This is a modulo q operation.

Openssl's BIGNUM library has been used to maintain all the large numbers and to perform modular and arbitrary precision arithmetic.

- **storeQAlphaBeta()**

int storeQAlphaBeta(BIGNUM * q, BIGNUM * alpha[], BIGNUM * beta[], char * uname)

This function is used to store q, Alpha[] and Beta[] values into a file <username>.qab on the disk to maintain them for next login session.The values are stored to the file after converting from the BIGNUM to decimal.

- **loadQab()**

unsigned char * loadQab(char * username)

This function is used to load the content of the file <username>.qab into the heap memory of the login process in form of a character string.

- **createEncryptedUserHistory()**

int createEncryptedUserHistory(char * historyFile, BIGNUM * hpwd)

This function is used to create a padded and encrypted constant size history file "<username>.history" on the disk. This size is initialized to 1048 for every user. This file will always be of the same size defined by historySize. This is called only at the time of initialization. This file is encrypted using a randomly generated hardened password hpwd. A text string is written to this to verify the correct file decryption at the login time.

- **encryptAndStoreUserHistory()**

int encryptAndStoreUserHistory(char * historyFile, BIGNUM * hpwd, unsigned char * userHistory)

This function takes in the history file name, hardened password hpwd and userHistory as the parameters. It is basically used to overwrite the already existing history file with the updated and encrypted user history. The user history is padded to ensure that the stored file is always of the same length.

- **decryptUserHistory()**

unsigned char * decryptUserHistory(char * historyFile, BIGNUM * hpwd)

This function takes in the history file and hardened password as the parameters. It is used to read the content of the user history file stored on the disk into a char string. This cipher string is then decrypted to recover the user history using the hardened password which will have max 8 lists of feature values corresponding to the last 8 successful logins along with string to verify correct decryption and padding.

- **registerAndInitialize()**

int registerAndInitialize(char * historyFile, char * newPassword, char * uname)

This function is called once to register each new user. Registration includes generation of hardened password and q for the user as well as creation of encrypted binary history file with no feature values initially. In addition, alpha and beta values are generated using user supplied password which are stored on the disk for future login verification.

- **generateXY()**

int generateXY(BIGNUM * alpha[], BIGNUM * beta [], BIGNUM * xValues[], BIGNUM * yValues[], BIGNUM * q, int inPhi[], const char * inPassword)

This function is used to recover 15 (X,Y) pairs using Alpha[] and Beta[]. The choice of Alpha and Beta as well as X value depends on the input feature values for the current login as explained in [1].

- **recoverHpwd()**

int recoverHpwd(BIGNUM * xValues[], BIGNUM * yValues[], BIGNUM * q, BIGNUM * recoveredHpwd)

This function is used to recover the hardened password using the 15 (X,Y) pairs calculated using alpha[] and beta[]. Lagrange interpolation (as explained in [1]) is used to recover the hardened password.

- **computeMeanAndVarianceOfUserHistory()**

int computeMeanAndVarianceOfUserHistory(std::string cppHistoryString, float mean[], float deviation[])

This function is used to compute mean and standard deviation for each feature using the 8 lists of feature values present in the user history.

- **randomizeAlphaBeta()**

int randomizeAlphaBeta(BIGNUM * q, BIGNUM * alpha[], BIGNUM * beta[], float mean[], float deviation[])

This function takes 160 bit q, alpha, beta , mean and deviation as the parameters. The function is used to insert random values in alpha[] and beta[] for the distinguishing features using the criteria mentioned in class. Correct values are maintained for both alpha and beta corresponding to non-distinguishing features.

- **updateInstructonTable()**

int updateInstructionTable(BIGNUM * q, BIGNUM * recoveredHpwd, std::string cppHistoryString, char * password, char * uname)

This function is used to update the instruction table after each successful login attempt. To do so, it uses the 160 bit randomly generated q and recovered hpwd to generate a new polynomial and new values for alpha[] and beta[] using the functions mentioned earlier. If the user history has 8 entries for the feature values corresponding to the last 8 successful login attempts, it computes the mean and standard deviation for each feature and randomize the alpha[] and beta[] values accordingly. After which, the new alpha[] and beta[] values are stored along with q to the disk.

- **postAuthenticationManager()**

int postAuthenticationManager(BIGNUM * q, BIGNUM * recoveredHpwd, char * recoveredUserHistory, int inPhi[], char * historyFile, char * password, char * uname)

This function inserts the new feature values corresponding to the current login into the user history and ensures that fixed history file size is maintained. This updated history is encrypted with a newly generated hpwd and stored to the disk. Finally, the instruction table is updated by calling updateInstructionTable() function.

- **authenticateLoginRequest()**

int authenticateLoginRequest(char * username, char * inPassword, int inPhi[], char * historyFile)

This function is used to authenticate the login request. It reads the values for q, alpha[] and beta[] from the qab file stored on the disk, recovers 15 (X,Y) pairs by calling generateXY(), recovers hpwd' by calling recoverHpwd(), decrypts the user's history file using hpwd' and verifies the decryption by searching for the success identifier. On successful login, it calls postAuthenticationManager() to perform post login operations which will help to authenticate next login attempt.

- **main()**

This is the main runner function which facilitates user interaction and calls the required function to authenticate the login request.

Experimentation and Analysis

Experiments have been performed with varied feature values to analyze the performance of the system implemented and to decide the best suitable value for the system parameter k. Only some of the results have been presented here to keep the report short.

❖ Threshold Values

Threshold values depend on various factors. The number of users who are going to use the same system with this authentication scheme and the variation in their typing behavior may govern the suitable threshold values. In addition, the measurement unit also decides the exact threshold values i.e., whether we are measuring the time in seconds or milliseconds or microseconds etc. Therefore, suitable values should be chosen at the time of system installation. In the paper [1], authors have mentioned the range for threshold values which they used for experimentation. The range for key press duration is [80, 125] ms and [70, 140] ms for latency.
We take the mid values of these ranges to set the threshold values. Therefore, the threshold vector used in the experimentation is:

int threshold[] = {100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100};

❖ Basic System Verification - Fast Typing Pattern (False Positives)

- **Sample Feature Vectors**

70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120

- **Result**
  The system is able to identify that the last feature vector is not the normal pattern for this user and thus, denies the login request. Same behavior has been observed for k from 0.1 to 10.

- **Analysis**
  If the user has fast typing speed(always below threshold values), any login request with slow typing pattern would be denied as it doesn't match the benign user's behavior.
  This concludes that if the user has a constant typing speed, the deviation will be zero and thus, increasing k won't have any impact.

❖ **Basic System Verification - Slow Typing Pattern (False Positives)**

- **Sample Feature Vectors**

120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
60 60 60 60 60 60 60 60 60 60 60 60 60 60 60

- **Result**
  The system is able to identify that the last feature vector is not the normal pattern for this user and thus, denies the login request. Same behavior has been observed for k from 0.1 to 10.

- **Analysis**
  If the user has slow typing speed (always above threshold values), any login request with fast typing pattern would be denied as it doesn't match the benign user's behavior.
  This concludes that if the user has a constant typing speed, the deviation will be zero and thus, increasing k won't have any impact.

❖ **Basic System Verification - False Negative**

- **Sample Feature Vectors**

70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
80 60 80 60 70 50 70 50 85 65 70 50 80 60 80
90 80 100 50 70 50 70 80 70 50 70 60 90 50 70
75 50 70 40 88 60 70 50 70 50 70 80 90 70 100
100 50 70 40 70 50 70 50 90 70 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 70 50 70 50 70 50 70 50 70 50 70 50 70
70 50 90 70 80 50 70 50 70 50 70 50 80 70 95
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

- **Result**
  The system is able to identify that the last feature vector is also the normal pattern for this user and thus, allows the login request. Same behavior has been observed for k from 0.1 to 10.

- **Analysis**
  If the user has some variation in the feature values but the values remain below or near threshold values, the system is able to accept the feature values near threshold as the correct pattern.

- **Experimentation with k - Randomness near threshold**
  - **Sample Feature Vectors**

120 50 70 110 70 60 60 90 80 80 120 70 100 105 70
80 110 110 60 120 70 90 70 60 65 90 110 80 60 80
90 80 100 50 90 80 110 80 100 120 90 90 125 50 70
75 50 90 80 88 110 120 105 95 90 100 110 90 70 100
100 100 70 40 70 85 110 80 60 100 80 50 70 110 70
120 70 70 50 70 90 75 110 110 70 90 90 100 50 120
70 90 120 90 100 80 60 50 120 90 85 90 70 90 70
70 120 90 70 80 90 70 60 80 80 85 50 80 110 105
100 100 100 80 100 90 100 100 100 100 100 100 100 100 100

  - **Result**
    First 8 login requests get accepted if the input password is correct. The last login request gets allowed if k is set to 1 but it gets denied if k is set to 0.1, 0.4 or 0.6

  - **Analysis**
    Most of the feature values for the first 8 vectors are below or near threshold and some are above threshold. Therefore, the mean is much below the threshold with some standard deviation. With k=1, we can decrease the number of distinguishing features and thus, the system allows login. But, for k = 0.1 to 0.6, the requirement for correct feature values becomes more strict as the number of distinguishing features are more.

- **Experimentation with k - Is 0.1 suitable?**
  - **Sample Feature Vectors**

60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
95 95 95 95 95 95 95 95 95 95 95 95 95 95 95

  - **Result**
    First 8 login requests get accepted if the input password is correct. The last login request gets allowed if k is set to 1 but it gets denied if k is set to 0.1.

  - **Analysis**
    All the values in the feature vectors are below the threshold. The last feature vector can be a valid pattern for the user in case he is typing slightly slower. But, the request gets denied for k=0.1. It means that the request which is intuitively correct is not acceptable to the system with k set to 0.1. Therefore, we can say that 0.1 makes it too strict. It may help to reduce false positives but it certainly increasing the number of false negatives.

- **Experimentation with k - Let's look for a better option**
  - **Sample Feature Vectors**

```
60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
```

  - **Result**
    First 8 login requests get accepted if the input password is correct. The last login request gets allowed if k is set to [0.4, 1] but it gets denied if k is set to 0.1.

  - **Analysis**
    Looking at the first 8 feature vectors, we can intuitively say that the last login request should be accepted. But, it gets denied for k = 0.1. The system behaves as expected for k = [0.4, 1]. Therefore, we can say that k = 0.4 may be a good option.

- **Experimentation with k - Verifying the learning**
  - **Sample Feature Vectors**

```
60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110
105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
```

  - **Result**
    First 8 login requests get accepted if the input password is correct. The last login request gets allowed if k is set to 0.6 but it gets denied if k is set to 0.4.

  - **Analysis**
    Looking at the first 8 feature vectors, we can't decide whether the last request should be allowed or not. It would depend on the system admin how much strict they want to make the system. Last login can be allowed if we want to decrease false negatives and thus, we can set k to 0.6. Higher values would also help to control the false negatives but they will result in increased false positives. We can set k to 0.4 if we want more secure system with some flexibility.

❖ Experimentation Result

The optimal value for k will depend on many factors. The most important is the balance between security and usability. So, it would depend on type of system for which this authentication system is used. If security is priority, k can be set to smaller values. But, if we want to balance both false positives and false negatives, k can be set to some moderate value.
From the above experiments, we feel that [0.4, 0.6] can be a good range for k. 0.4 would decrease the false positives as compared to 0.6 but may deny some of the benign requests. Therefore, the actual value should be chosen based on the security/usability requirement.

We have set k=0.5 for our code as it provide as balance maintaining sufficient security without causing much trouble to a genuine user.

The total number of users, the variation in their typing patterns and the entropy required also govern the selection of most optimal value for k.

❖ Verification of the Result - False Positives

- **Sample Feature Vectors**

  60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
  70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
  80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
  90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
  105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
  100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
  105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
  130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
  120 120 120 120 120 120 120 120 120 120 120 120 120 120 120

- **Result**
  First 8 login requests get accepted if the input password is correct. The last 2 login requests get denied for k = 0.6

- **Analysis**
  Looking at the first 8 feature vectors, we can say that the last 2 requests should be denied as the values are much higher. The system behaves as expected with k = 0.6.

- **Verification of the Result - Comparing k = 0.4, 0.5 and 0.6**
  - **Sample Feature Vectors**

```
60 70 80 90 60 70 80 90 60 70 80 90 60 70 80
70 80 90 60 70 80 90 60 70 80 90 60 70 80 90
80 90 60 70 80 90 60 70 80 90 60 70 80 90 60
90 60 70 80 90 60 70 80 90 60 70 80 90 60 70
105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
100 100 100 100 100 100 100 100 100 100 100 100 100 100 100
105 105 105 105 105 105 105 105 105 105 105 105 105 105 105
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110
120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
130 130 130 130 130 130 130 130 130 130 130 130 130 130 130
```

  - **Result**
    First 8 login requests get accepted if the input password is correct. The last 2 login requests get allowed for k = 0.6 but get denied for k = 0.4, 0.5.

  - **Analysis**
    This confirms that more security can be provided by setting k = 0.4/0.5. k=0.6 allows some flexibility and thus, may be preferred in some scenarios.

An exhaustive analysis can't be provided in this report. But, the experiments performed suggest that [0.4, 0.6] is a good range for k and most optimal value can be chosen keeping security and flexibility in mind.

Integration

This authentication system can be integrated with any new or already existing password based authentication system such as NFS etc. To facilitate integration to a real system, functionality to capture the actual keystroke timings should be implemented. Different modules can be used to record measurements available from keyboard to determine Dwell time (the time a key pressed) and Flight time (the time between "key up" and the next "key down"). The bio metric values so obtained can then be fed into the developed authentication code.The module developed would need some modification to accept the feature values from new module recording the measurements instead of reading from the file or the terminal.

The values for the threshold should be decided based on the type of users, the type of keyboards used and variation in typing pattern of the users using the system. This study should be carried out initially before deploying the system for actual use (beta testing can be performed). The value for k should also be decided based on the security requirements. If usability is of higher preference, higher values for k can be chosen. k = 0.6 is a good value to balance both security and usability. But, optimal value should be chosen as per the actual environment where system is going to be deployed.

Security Evaluation

Secure coding practices have been followed while developing the system. It has been ensured that the code doesn't use any unsafe functions and attention has been given to avoid software vulnerabilities. For example:

- Functions implementing maximum limit on length such as strnlen(), strncpy(), strncat() have been used to avoid buffer overflow conditions
- Length check has been performed wherever static character arrays have been used
- Use of scanf() has been avoided as it can lead to buffer overflow. fgets() has been used as it allows to enforce maximum length on input.
- Use of printf and other functions which can trigger format string vulnerability has been avoided.
- Proper bound checking and exception handling have been done.
- The memory storing big numbers, strings and other cryptographic data has been cleared after use.

In addition, the code has been evaluation with flaw finder to identify the vulnerabilities. No critical vulnerability was found in the analysis. Some hits were observed for use of static arrays. But, proper bound checking has been implemented in such cases and thus, it won't lead to any vulnerability in this context.

References

[1] Monrose et. al. Password hardening based on keystroke dynamics.